Tercer Trabajo Práctico

Junio de 2010 | Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Bianchi, Mariano	92/08	marianobianchi08@gmail.com
Brusco, Pablo	527/08	pablo.brusco@gmail.com
Di Pietro, Carlos Augusto Lyon	126/08	cdipietro@dc.uba.ar

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Índice

1. Ejercicio 1

1.1. Introducción

El problema a resolver en el presente trabajo práctico consiste en dado un grafo simple, encontrar un *MAX-CLIQUE* para dicho grafo. Una clique es un subgrafo completo del grafo original. Un *MAX-CLIQUE* , es una clique tal que no exista otra que contenga más vértices.

Este problema es muy conocido. Además, no está computacionalmente resuelto y tiene infinidad de aplicaciones en distintos problemas de la vida real, lo que hace que sea un importante objeto de estudio. Algunas de sus aplicaciones más estudiadas provienen de áreas como bioinformática, transporte, diseño de tuberías, diseño de redes energéticas, procesamiento de imágenes, seguridad informática, electrónica, etc.

1.2. Algunas aplicaciones

Una aplicación posible podría darse por ejemplo, en el contexto de una compañía de telefonía móvil. Como bien se conoce, este tipo de empresa ofrece un plan llamado "plan empresas" para el cuál todos los teléfonos que se encuentren bajo este, tienen la posibilidad de comunicarse entre sí de forma libre.

Para la empresa, podría ser de interés conocer algún grupo de personas que estén comunicados todos entre sí para ofrecerles un "plan empresas" y así beneficiarlos dándole la oportunidad de aumentar el caudal de llamadas entre sí, por un precio más razonable.

Podemos pensar el modelo de la siguiente manera:

- Vértices: Son los celulares de los clientes de la empresa de telefonía.
- Ejes o aristas: Existe una arista entre dos vértices (o teléfonos) A y B si alguna vez se realizó una llamada entre ambos (A llamó a B o vice versa)[1].

Con este modelo, encontrar una clique de tamaño K significa encontrar un grupo de K celulares que se hayan comunicado todos entre sí alguna vez durante un período de tiempo predeterminado. Pasa lo mismo si se busca un *MAX-CLIQUE* . Esto sería equivalente a encontrar el mayor grupo de personas que estén comunicados todos entre sí[2].

De la misma forma, se puede pensar al revés, y quizás a la empresa le interese conocer grupos de celulares que estén todos comunicados entre sí, para NO ofrecerles el "plan empresas" ya que de esa manera ese grupo de personas podría eventualmente bajar el consumo de sus llamadas descendiendo las ganancias de la empresa de telefonía.

[1] Sería conveniente elegir un período de tiempo acotado para ver si se produjo dicha llamada o no y así poder armar el grafo.

[2] Una variante sería buscar el *MAX-CLIQUE* durante un año todos los días y quedarse con aquellos que se haya repetido la mayor cantidad de veces.

2. Ejercicio 2

2.1. Introducción

En esta sección se presentara un algoritmo exacto para resolver el problema de encontrar la Clique Máxima en un grafo.

Aun no se conocen algoritmos buenos, es decir, polinomiales con respecto al tamaño de la entrada, para resolver este problema, asi que nos consentraremos en realizar mejoras al algoritmo de fuerza bruta que considera todos los casos.

2.2. Explicación

Un algoritmo de fuerza bruta para resolver el problema de Max-Clique podría simplemente intentar formar el conjunto más grande de nodos, donde ese conjunto sea completo, intentando todas las posibilidades eligiendo todos los conjuntos de un cierto tamaño, luego intentar con un tamaño menor, etc. Probablemente la complejidad de un algoritmo de este estilo sea n^n donde n es la cantidad de nodos del grafo.

Una mejora que surge casi inmediatamente es utilizar la técnica de BackTracking, cuya función principal es intentar podar el arbol implicito de combinaciones posibles.

De todas maneras, implementar solo un BackTracking parece ser poco con respecto a las mejoras que se pueden lograr. A continuación, se explicara el algoritmo implementado con un pseudocodigo y se verá cada una de las mejoras por separado. **Algoritmo Exacto**(G: grafo)

```
CliqueMayor = vacia
componentesConexas = DetectarComponentesConexas(G)
Para cada componente en componentesConexas {
        heap = crearHeap(G,componentes)
        Mientras noVacio(heap) ∧ top(heap) ≥ tam(CliqueMayorActual){
            v = top(heap)
            Para tamCliqueABuscar desde grado(v)+1 hasta
tam(CliqueMayorActual)+1{
                vecinosFiltrados = filtrarVecinosMenores(v, tamCliqueABuscar-1)
                Si tam(vecinosFiltrados) +1 ≥ tamCliqueABuscar
                    temp = BuscoCliqueDeTamañoK(tamCliqueABuscar,
vecinosFiltrados)
                    Si tam(CliqueMayorActual) < tam(temp)
                        CliqueMayorActual = temp
            }
        }
```

Algoritmo 1: Pseudocódigo del algoritmo exacto

Primero, se detectan las componentes conexas del grafo, ya que buscar la max clique en todo el grafo es equivalente a quedarse con la máxima de las max cliques de cada componente conexa.

Luego, para cada componente, se crea un heap que contiene los nodos de la componente ordenados por mayor grado. Esto no parece tener mucha importancia, pero se aclarará a medida que se avanza con la explicación.

3. Ejercicio 3

3.1. Introducción

Para la resolución de este ejercicio se debía desarrollar e implementar una heurística constructiva para resolver el problema de encontrar un *MAX-CLIQUE* dado un grafo simple.

3.2. Explicación

En una primera aproximación al problema, se pensó un algoritmo bastante sencillo. La idea del mismo radicaba en ir tomando los nodos en orden de grados, es decir, comenzando con los de mayor grado hasta llegar a los de menor grado. De esta forma, uno puede pensar que al tomar primero los vértices de mayor grado, hay mas chances de encontrar una clique de mayor tamaño.

Esto es claramente una heurística válida que utiliza la técnica de algoritmo goloso. Pero es claro también que se pueden encontrar fácilmente ejemplos de grafos en los que dicho algoritmo funcione tan mal como uno quiera.

A fines de evitar en cierto grado muchos casos para los cuales este algoritmo funciona mal, se planteó uno nuevo que utiliza la misma idea pero que la misma no se realiza sobre todos los nodos del grafo sino que se hace sobre un subconjunto de los mismos. Para formar dicho conjunto, se implementó un algoritmo que revisa todas las combinaciones de 2 vértices distintos (siempre y cuándo haya 2 o más vertices en el grafo) que sean vecinos entre sí y se guarda en un conjunto de vértices aquellos que sean vecinos a ambos vértices y además se guardan los 2 vértices en cuestión. Esto se repite para cada posible combinación de vértices de a 2, guardando siempre el conjunto más grande que se haya encontrado completado de la forma antes mencionada.

Una vez encontrado este subgrafo, se procede a realizar el algoritmo goloso antes mencionado pero sobre dicho subgrafo, es decir, se busca el nodo con mayor grado en ese subgrafo y se coloca en un conjunto, el cuál será devuelto como clique al terminar el algoritmo. Luego, para el resto de los nodos del subgrafo, se va tomando de a uno a la vez en orden de mayor a menor grado (considerando sólo los adyacentes que pertenecen al subgrafo) y se verifica que sea adyacente a todos los que pertenecen a la clique. Si lo es, entonces se lo inserta en el conjunto sino se lo descarta. Finalmente, se prosigue con estos pasos hasta haber intentado con todos los nodos del subgrafo devolviendo entonces la clique encontrada.

A continuación se adjunta el pseudocódigo del algoritmo constructivo antes descripto y el de las funciones auxiliares pertinentes. En los mismos, utilizaremos a “n” como forma de expresar la cantidad de nodos pertenecientes al grafo.

AlgoritmoConstructivo(G: grafo)

```
1  frontera = mayorFronteraEnComun(G) ;        // O(n^3)
2  res = ∅;                                    // O(1)
3  mientras frontera ≠ ∅
4      v = elMasRelacionado(G,frontera);       // O(n^2)
5      si esVecinoDeTodos(G,res,v);            // O(n)
6          insertar v en res;                  // O(log(n))
7      fin si
8      eliminar v de frontera;                 // O(log(n))
9  fin mientras
10 si res == ∅;                                // O(1)
11     insertar v_0 en res;                    // O(1)
12 fin si
13 devolver res
```

Algoritmo 2: Pseudocódigo del algoritmo constructivo

mayorFronteraEnComun(G: grafo)

```
1  aux = ∅;                                                      // O(1)
2  res = ∅;                                                      // O(1)
3  paratodo u,v ∈ V_G tq (u,v) ∈ X_G
4      aux = (adyacentes(G,u) ∩ adyacentes(G,v)) ∪ u ∪ v;        // O(n)
5      si #aux > #res;                                           // O(1)
6          res = aux;                                            // O(1)
7      fin si
8  fin paratodo
9  devolver res
```

Algoritmo 3: Pseudocódigo de un algoritmo secundario al constructivo

El pseudocódigo de las funciones *elMasRelacionado, esVecinoDeTodos* y *adyacentes* no se detalla ya que no son algoritmos de gran complejidad. Igualmente, se explicarán brevemente a continuación.
En el caso de *elMasRelacionado*, recibe dos parámetros, un grafo y un conjunto de vértices. Esta función devuelve el vértice de mayor grado del grafo inducido por los vértices de ese conjunto. La función *esVecinoDeTodos* recibe un grafo, un vértice y un conjunto de vértices. Devuelve verdadero si y sólo si el vértice es adyacente a todos los vértices del conjunto. Por último, en el caso de la función *adyacentes*, recibe un grafo y un vértice como parámetros y devuelve el conjunto de todos los vértices adyacentes al pasado como parámetro.

3.3. Análisis de la complejidad del algoritmo

Para realizar el siguiente análisis de complejidad vamos a remitirnos al pseudocódigo de la función *AlgoritmoConstructivo* adjunto en la sección anterior. Cabe recordar que cada vez que se haga mención a "n" nos estaremos refiriendo a la cantidad de nodos del grafo al que se quiere analizar.
En la primer línea se realiza una llamada a la función *mayorFronteraEnComun* y una asignación. Estas 2 operaciones tienen una complejidad de $O(n^3)$. Más adelante se explicará el por qué de la misma. Luego, dentro del ciclo, la función que mayor complejidad temporal tiene es

elMasRelacionado y la misma es $O(n^2)$. Esta complejidad se debe a que para cada vértice del grafo inducido por el conjunto de vértices pasado como parámetro, se debe calcular cuántos vecinos tiene dentro de ese grafo, para lo que se debe recorrer todo el conjunto una vez por cada vértice. Como a lo sumo puede haber "n" vértices en ese conjunto, debo recorrerlo "n" veces por cada vértice. Entonces, su complejidad es $O(n^2)$.
Deberiamos ver ahora cuántas veces va a ejecutarse el ciclo. Este finalizará una vez que el conjunto llamado "frontera" quede vacío. Este arranca con el valor de la mayor frontera en común, por lo que a lo sumo puede ser de tamaño "n", es decir, todos los vértices pueden pertenecer a él en el peor caso. Podemos observar además, que en cada paso del ciclo, su tamaño disminuye en uno (línea 8, al eliminar un vértice del conjunto) , por lo que a lo sumo el ciclo se ejecutará "n" veces. Como la función más compleja dentro del ciclo era *elMasRelacionado* con una complejidad de $O(n^2)$ y la misma se ejecutará "n" veces, entonces la complejidad total temporal del ciclo es de $O(n^3)$. Por lo tanto, la complejidad del algoritmo constructivo es de $O(n^3)$.

Para completar este análisis, falta justificar por qué la función *mayorFronteraEnComun* tiene complejidad $O(n^3)$.

3.4. Detalles de implementación

3.5. Resultados

3.6. Debate

3.7. Conclusiones

4. Ejercicio 4

5. Anexos